# Basics of CPU Pipelining

Prof. Naga Kandasamy
ECE Department
Drexel University

April 11, 2016

These notes are derived from:

- D. Patterson and J. Hennessy, *Computer Organization and Design*, $4^{\text{th}}$ Edition, Morgan Kaufmann, 2008.
- J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, $5^{\text{th}}$ Edition, Morgan Kaufmann, 2011.

## 1 Basic Concepts

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. A pipeline is similar in concept to an assembly line: each step of the pipeline completes one part of the instruction. Each of these steps is called a *pipe stage* or a *pipe segment* and the stages are connected one to the next to form a pipe. Instructions enter at one end, are processed through the stages, and exit at the other end. Figure 1 shows the MIPS pipeline comprising of five stages where each stage corresponds to one step in the execution of an instruction. Recall that all MIPS instructions take the following five steps:

- Fetch instruction from memory (the IF stage).

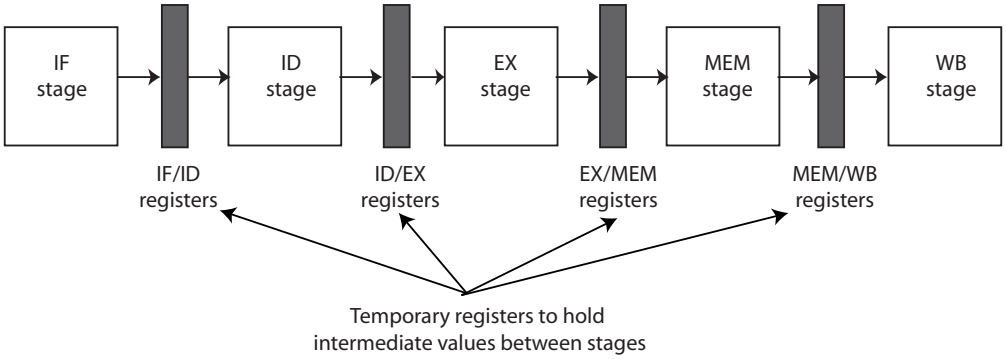


Figure 1: Structure of the MIPS pipeline.

| Instruction number | Clock number 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Instruction $i$ | IF | ID | EX | MEM | WB | | | | |
| Instruction $i$+1 | | IF | ID | EX | MEM | WB | | | |
| Instruction $i$+2 | | | IF | ID | EX | MEM | WB | | |
| Instruction $i$+3 | | | | IF | ID | EX | MEM | WB | |
| Instruction $i$+4 | | | | | IF | ID | EX | MEM | WB |

Figure 2: The simple MIPS pipeline in operation.

- Read registers while decoding the instruction (the ID stage). Note that the MIPS instruction format allows the reading of registers and the decoding to occur simultaneously. If these register values are not required by the instruction, they are simply discarded later in the pipeline.
- Execute the instruction (in the case of R-type instruction) or calculate an address (in the case of I-type instruction). This is done in the EX stage.
- Access an operand in data memory (the MEM stage).
- Write the result into a register in the register file. This is done in the WB stage.

It is important to note that not all instructions require the five steps described above. For example, R-type instructions such as `add` and `sub` do not require the MEM step, and the store instruction `sw` does not require the WB step. However, every instruction progress through all pipeline stages—if a particular stage is not needed for an instruction, that instruction proceeds through the stage without doing anything.

The throughput of the pipeline is determined by how often an instruction exits the pipeline. The time required between moving an instruction one step down the pipeline is a clock cycle, and the length of the clock cycle is determined by the time required for the slowest pipeline stage (because all stages proceed at the same time). The goal of the pipeline designer is to balance the length of the pipeline stages. If the stages are perfectly balanced, then the time per instruction on the pipelined machine—assuming ideal conditions (no stalls)—is equal to

$$\frac{\text{Time per instruction on nonpipelined machine}}{\text{Number of pipeline stages}}$$

Under these conditions, the speedup from pipelining equals the number of pipe stages. However, usually, the stages will not be perfectly balanced. Moreover, pipelining incurs some overhead—for example, the delay through the temporary registers/latches used to store the intermediate values between stages.

Figure 2 shows the simple pipeline in operation. On each clock cycle, another instruction is fetched and begins its five step execution. If an instruction is started every clock cycle, the performance will be five times that of a machine that is not pipelined. Note that pipelining increases the CPU instruction throughput—the number of instructions completed per unit time—but does not reduce the execution time of an individual instruction. In fact, the execution time of each individual instruction might increase slightly due to pipelining overhead.

# 2 Making the Pipeline Work

We will now examine some implementation issues related to pipelining and show how to deal with these problems.

First, we have to determine what happens on every clock cycle and make sure that the overlapping instructions have sufficient resources to proceed with their execution. For example, a single ALU cannot be asked to compute both a memory operand address and perform an addition at the same time. Since every pipe stage is active on every clock cycle, this requires that all operations in a pipe stage complete in one clock cycle and any combination of operations performed by multiple instructions able to occur during a clock cycle.

- The PC must be incremented on each clock cycle. This must be done in the IF stage rather than ID. This will require an additional incrementer, since the ALU in the EX stage is already busy on every cycle and cannot be used to increment the PC.
- A new instruction must be fetched on every clock cycle—this is also done in IF.
- A new data word is needed on every clock cycle—this is done in MEM.
- There must be a separate memory buffer register for load and store instructions since these instructions may overlap in time.
- Additional latches are needed to hold values (the IR, the ALU output, next PC, etc) that are needed later in the pipeline, but may be modified by a subsequent instruction.

The biggest impact of pipelining on processor resources is in the memory system. With the same number of steps per instruction, two memory accesses are required per clock cycle in a pipelined design, whereas in a non-pipelined machine, two memory accesses are required every five clock cycles. Therefore, the peak memory bandwidth must be increased by five times over the non-pipelined design. During the EX stage, the ALU can be used for three different functions: to calculate the memory address of an operand (for `lw` and `sw` instructions), a branch address calculation (for `beq` and `bne` instructions), or an ALU operation (for `add` and `sub` instructions). A MIPS instruction performs at most of these functions, and so no conflict arises.

# 3 Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These situations are called *hazards*, and there are three classes of hazards.

- *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
- *Data hazards* arise when an instruction depends on the results of a previous instruction still in the pipeline.
- *Control hazards* arise from the pipelining of branches and other instructions that change the program counter.

Hazards in pipelines can make it necessary to stall the pipeline. A *stall* in a pipelined processor requires that some instructions be allowed to proceed, while others are delayed. Typically, when

an instruction is stalled, all instructions later in the pipeline than the stalled instruction are also stalled. Instructions earlier than the stalled instruction can continue, but no new instructions are fetched during the stall.

A stall causes the pipeline performance to degrade from the ideal performance. We know that

$$\begin{aligned}\text{Pipeline speedup} &= \frac{\text{Average instruction time without pipeline}}{\text{Average instruction time with pipeline}} \\ &= \frac{\text{CPI without pipelining} \times \text{Clock cycle without pipelining}}{\text{CPI with pipelining} \times \text{Clock cycle with pipelining}} \\ &= \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} \times \frac{\text{CPI without pipelining}}{\text{CPI with pipelining}}\end{aligned}$$

The ideal CPI on a pipelined machine is

$$\text{Ideal CPI} = \frac{\text{CPI without pipelining}}{\text{Pipeline depth (or number of pipeline stages)}}.$$

Substituting the above equation into the speedup equation gives us:

$$\text{Speedup} = \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} \times \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{CPI with pipelining}}.$$

If we consider pipeline stalls, then

$$\text{CPI with pipelining} = \text{Ideal CPI} + \text{Pipeline Stall clock cycles per instruction}$$

We can substitute this in the above equation and obtain,

$$\text{Speedup} = \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} \times \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI + Pipeline Stall cycles}}$$

Ignoring the small increase in clock cycle length due to the pipelining overhead, we obtain a simpler formula for speedup as

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI + Pipeline Stall cycles}}$$

# 4 Structural Hazards

When a processor is pipelined, the overlapped execution of instructions requires duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions cannot be accommodated due to resource conflicts, the processor is said to have a structural hazard. For example, assume that a processor has single-ported memory unit to store both instructions and data. When an instruction needs to access memory, the pipeline must stall for one clock cycle since the processor cannot fetch the next instruction because the memory-data reference is using the memory port. Figure 3 shows this scenario for a load instruction. With only one memory port, the pipeline cannot initiate a data fetch and instruction fetch in the same clock cycle. A load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall. No instruction is fetched on clock cycle 4. All other instructions in the pipeline proceed normally.

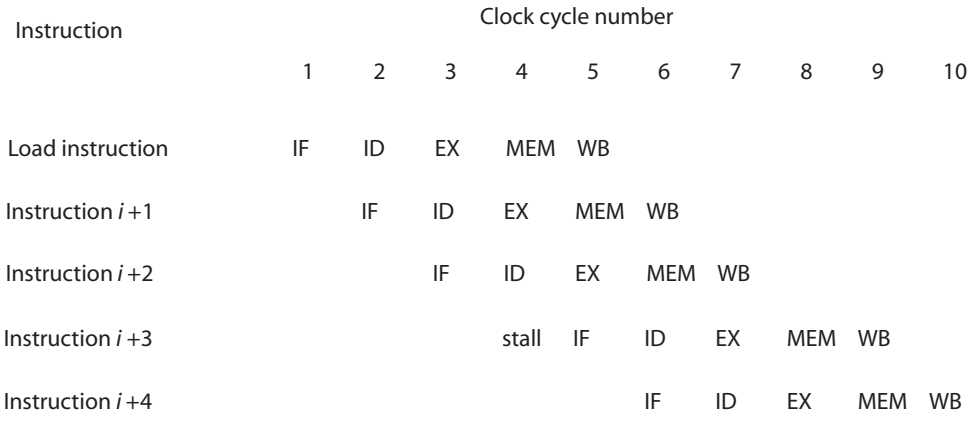


Figure 3: A pipeline stalled for a structural hazard—load with one memory port.

# 5 Data Hazards

Since pipelining overlaps the execution of multiple instructions, it introduces data and control hazards. Consider the pipelined execution of these instructions

```
add $s1, $s2, $s3
sub $s4, $s1, $s5
```

The `sub` instruction has a source register $s1 that is the destination of the `add` instruction. The `add` instruction updates $s1 at the end of the WB stage whereas `sub` will read $s1 in the ID stage. This problem is called a data hazard, and unless precautions are taken, `sub` will read the wrong value and try to use it. One option is to stall the pipeline after fetching `sub` until `add` is finished with the WB stage. This, however, introduces stall cycles and reduces the pipeline throughput. The problem posed in the above example can be solved with a simple hardware technique called *forwarding*—also called *bypassing*. This method works as follows: The ALU result is always fed back to the ALU input latches. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, the control logic selects the forwarded result as the ALU input rather than the value read from the register file. Note that with forwarding, if `sub` is stalled for some reason, the `add` instruction will complete and write the register file, and the bypass will not be activated, causing the value from the register file to be used.

In the MIPS pipeline shown in Figure 1, results must be forwarded not only to the instruction that immediately follows, but also to the instruction after that. Since the WB and ID stages of the first and third instructions overlap, we must continue to forward the result. Figure 4 shows a set of instructions in the pipeline and the forwarding operations that must occur.

Each level of bypass requires a buffer and a pair of comparators to examine whether the adjacent instructions share a destination and a source. Figure 5 shows the structure of the ALU and its bypass unit as well as what values are in the bypass registers for the instruction sequence shown in Figure 4. The contents of the buffer are shown at the point where the `and` instruction of the code sequence in Figure 4 is about to begin its EX stage. The `add` instruction that computed the register $s1 is in its WB stage. The forwarding logic (not shown in the figure) tests if either of the two previous instructions (`add` and `sub`) wrote a register that is the input to the `and` instruction.

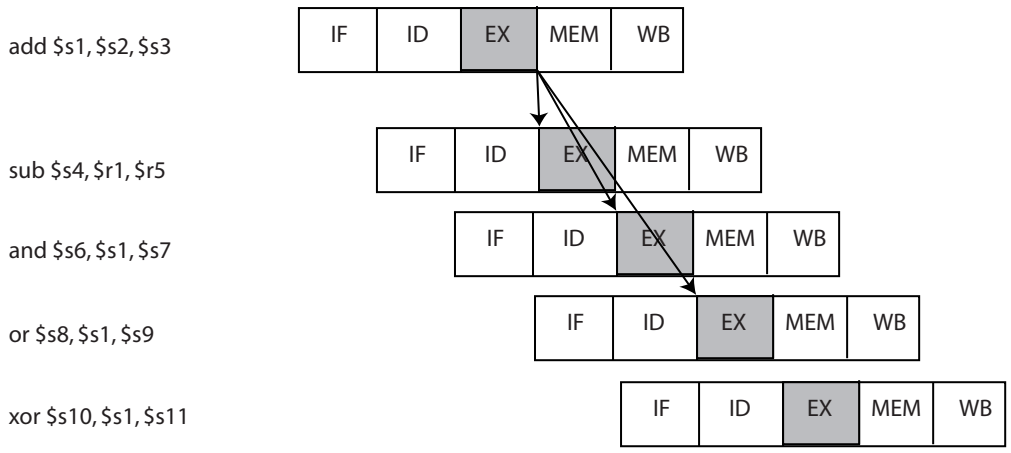


Figure 4: A set of instructions that need results forwarded to them within the pipeline. The `add` instruction sets register $s1 and the next four instructions use it. So, the value of $s1 must be forwarded to the `sub`, `and`, and `or` instructions. By the time the `xor` instruction reads $s1 in its ID phase, the `add` instruction has already finished WB, and the most recent value of $s1 is available from the register file.

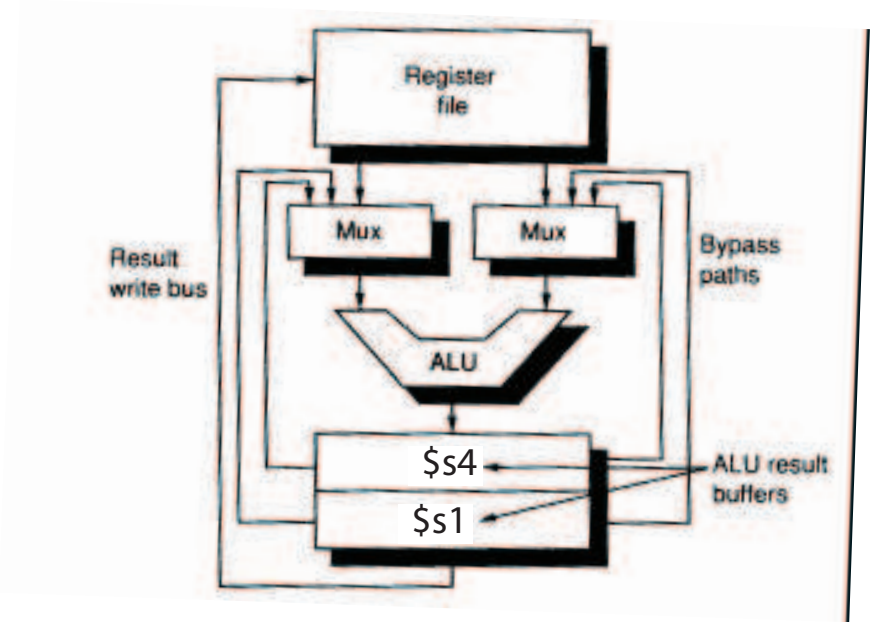


Figure 5: The ALU and its bypass unit in the EX stage of the pipeline.

Then, the left input multiplexor is set to pass the just-computed value of $s1 from the bypass register rather than the value read from the register file as the first operand to the `and` instruction. The result of the `sub` instruction, $s4, is in the first buffer.

Forwarding can be generalized to include passing a result generated by a stage directly to any other stage that requires it. For example, consider the following sequence.

```
add $s1, $s2, $s3
sw  $s1, 10($s1)
```

To prevent a pipeline stall in this sequence, we must forward the value of $s1 from the ALU both to the ALU (so that it can be used for address calculation by the `sw` instruction) and to the MEM stage where it can be placed in the memory buffer register (MBR) on its way to main memory.

The load instruction (`lw`) has a delay or latency that cannot be eliminated by forwarding alone. For example, consider the code sequence shown in Figure 6 assuming that forwarding is enabled between the EX and MEM stages. The `add`, `sub`, and `and` instructions are dependent on the value of register $s1 that is loaded from memory. The `add` instruction cannot execute in cycle 5 since `lw` loads the data from memory only at the end of its MEM cycle. A pipeline interlock detects this hazard and stalls the pipeline beginning with the instruction that wants to use the data until the sourcing instruction produces it. This delay cycle, called a *pipeline stall*, allows `lw` to load data from memory, and once the data arrives from memory, the value can be forwarded from the MEM stage to the EX stage for the `add` and `sub` instructions. When the `and` instruction enters its ID stage, the load instruction has completed WB, and therefore, `and` can simply read the value of $s1 from the register file.

In the pipeline shown in Figure 1, all hazards can be checked during the ID phase of the pipeline. If a data hazard exists (in spite of forwarding between the stages), the instruction is stalled until the hazard is resolved.

Many types of stalls are quite frequent. For example, the typical code generation pattern and its pipelined execution for a statement such as $a = b + c$ is shown in Figure 7. Here, the `add` instruction must be stalled to allow the load of `c` to complete. The `sw` instruction need not be delayed since the EX stage can forward the ALU result directly to the MEM stage for storing to memory.

Rather than just allow the pipeline to stall, the compiler could try to schedule the pipeline to avoid these stalls by reordering or rearranging instructions in a code sequence. For example, the compiler could try to avoid generating code with a load followed by an immediate use of the load destination register. This technique is called pipeline scheduling or instruction scheduling. Consider the following high-level code

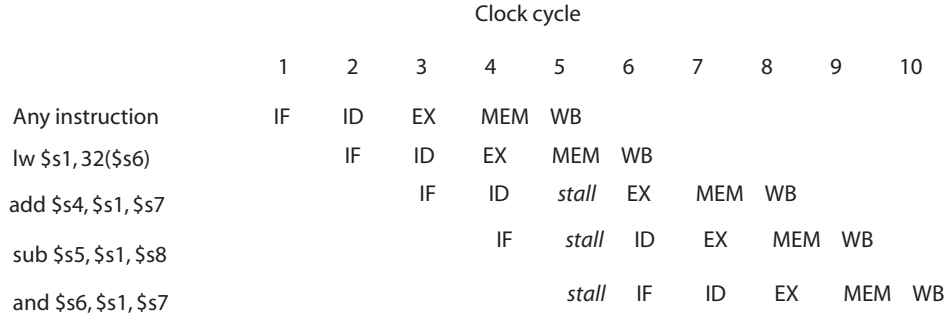


Figure 6: Pipeline hazard occurring when the result of a load instruction is used by the next instruction as a source operand and is forwarded. The value is available when `lw` returns from memory at the end of its MEM stage. However, this value is needed by the `add` instruction at the start of its EX stage. So, `add` and the subsequent instructions are stalled until the data is available.

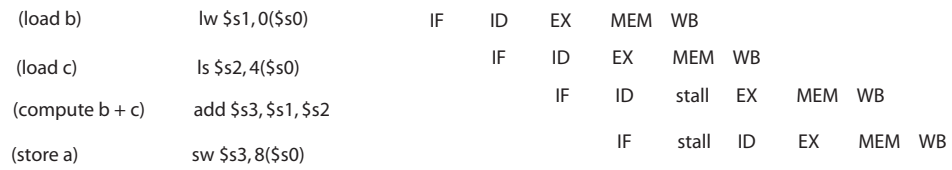


Figure 7: The pipelined code sequence for $a = b + c$.

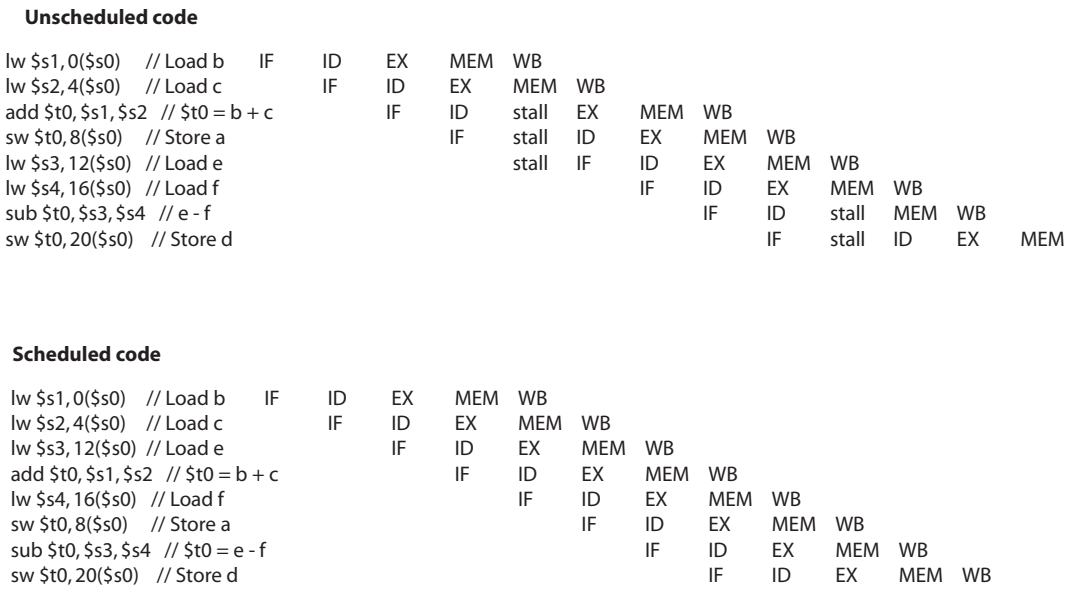


Figure 8: The unscheduled code and the scheduled code that avoids pipeline stalls.

```
a = b + c;
d = e - f;
```

Figure 8 shows both the unscheduled and scheduled code generated by the compiler. The unscheduled code will result in two pipeline stalls since the two load statements are followed by an immediate use of the load destination register. The scheduled code reorders instructions in the original code to eliminate pipeline stalls. There is a dependence between the ALU instruction and the store, but the pipeline structure allows the result to be forwarded. Also, notice that the use of different registers for the first and second statements was critical for the scheduled code to be legal. In particular, if the variable `e` were loaded in the same register as `b` or `c`, the scheduled code would not be correct. So, in general pipeline scheduling can increase the number of registers required.

# 6 Control Hazards

Control hazards, if not handled efficiently, can cause greater performance loss for the pipeline than data hazards. When a branch is executed, it may or may not change the program counter (PC) to something other than its current value plus 4. If a branch changes the PC to its target address, it is a *taken* branch. If it falls through, it is *not taken* (or *untaken*).

The penalty incurred by a branch statement in terms of stall cycles can be reduced by:

- Finding out (or resolving) whether the branch is taken or not taken earlier in the pipeline.

- Computing the taken PC (address of the branch target) earlier in the pipeline.

Both steps should be taken as early in the pipeline as possible.

In the MIPS machine, the branch instructions, `beq` and `bne`, require testing only equality to zero. Since we read the value of the registers from the register file in the ID stage, it is possible to resolve the branch by the end of the ID cycle using special logic devoted to this test (e.g., a comparator in the ID stage). Computing the branch target address requires a separate adder, which can add during ID (since the ALU in the EX stage may be servicing some other instruction). With the separate adder and a branch decision made during ID, there is only a one-cycle stall on branches (assuming we stall the pipeline until the branch is resolved).

## Reducing Pipeline Branch Penalties

There are several methods for dealing with pipeline stalls due to branches, and we will briefly discuss three simple compile-time schemes. In these schemes, the predictions are static, that is, they are fixed for each branch throughout the program execution and the predictions are compile-time guesses.

- *Stall the pipeline.* The easiest scheme is to freeze or stall the pipeline, preventing any instructions after the branch from entering the pipeline until the branch destination is known. In the MIPS pipeline where the branch is resolved in ID, this scheme will incur a one-cycle stall penalty for each branch instruction.
- *Predict the branch as not taken.* A slightly more complex scheme is to predict the branch as not taken, simply allowing the pipeline to continue as if the branch were not executed. Care must be taken not to change the processor state (memory values or register values in the register file) until the branch outcome is definitely known. In the MIPS pipeline, the *predict-not-taken* scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction. If the branch is taken, however, we need to stop the pipeline and

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Untaken branch instruction | IF | **ID** | EX | MEM | WB | | | | |
| Instruction *i*+1 | | **IF** | ID | EX | MEM | WB | | | |
| Instruction *i*+2 | | | IF | ID | EX | MEM | WB | | |
| Instruction *i*+3 | | | | IF | ID | EX | MEM | WB | |
| Instruction *i*+4 | | | | | IF | ID | EX | MEM | WB |
| Taken branch instruction | IF | **ID** | EX | MEM | WB | | | | |
| Instruction *i*+1 | | **IF** | **IF** | ID | EX | MEM | WB | | |
| Instruction *i*+2 | | | *stall* | IF | ID | EX | MEM | WB | |
| Instruction *i*+3 | | | | *stall* | IF | ID | EX | MEM | WB |
| Instruction *i*+4 | | | | | *stall* | IF | ID | EX | MEM |

Figure 9: The predict-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, as determined in ID, we have fetched the fall-through instruction and we just continue. If, however, the branch is taken during ID, we restart the fetch at the branch target, causing all instructions following the branch to stall one clock cycle.

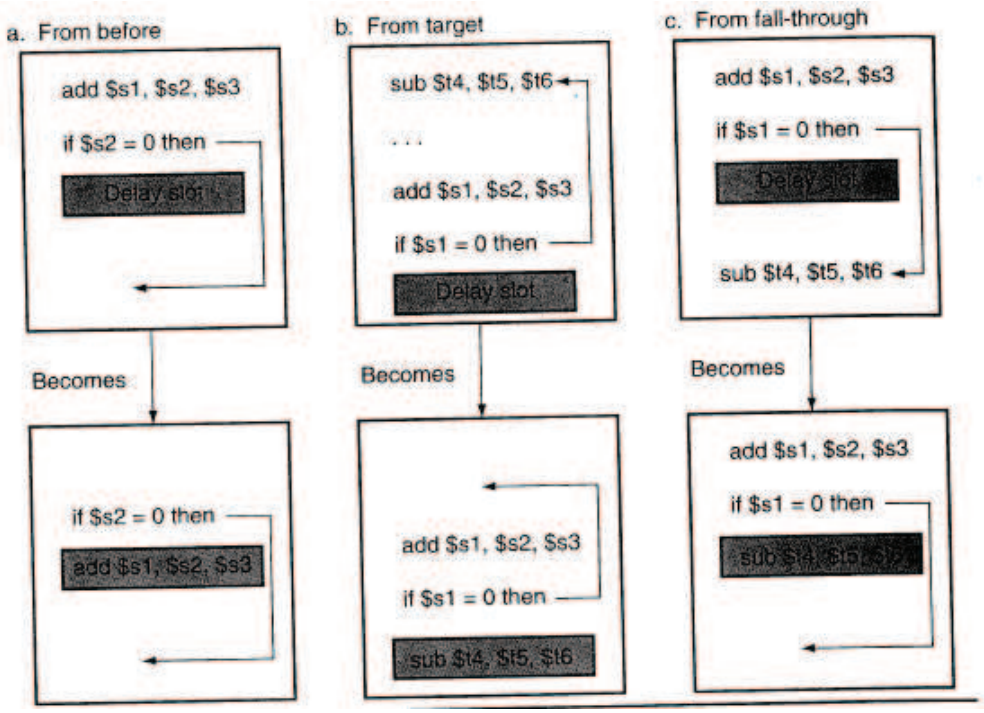


Figure 10: Scheduling the branch delay slot. The top box in each pair show the code before scheduling and the bottom box shows the scheduled code. In strategy (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code snippets in (b) and (c), the use of $s1 in the branch instruction prevents the `add` (whose destination register is $s1) from being moved into the branch delay slot. In (b), the delay slot is scheduled from the branch target; usually the target instruction will need to be copied since it can be reached via another path. Strategy (b) is preferred when the branch is taken with high probability (such as a loop branch). Finally, as shown in strategy (c), the delay slot can be scheduled using an instruction from the fall through or not taken path. To make the schedules in (b) and (c) legal, it must be OK to execute the `sub` instruction even when the branch goes in the unexpected direction. By "OK", we mean that the work done by `sub` is wasted but the program still executes correctly. This is the case, for example, if in strategy (b), $t4 were an unused temporary register in the not-taken path.

restart the IF cycle. Figure 9 shows both scenarios.

- *Delayed branches.* A delayed branch always executes the following instruction, but the second instruction following the branch will be affected by the branch. The compiler will try to place an instruction that always executes after the branch in the *branch delay slot*. The job of the compiler is to make the successor instructions valid and useful. Figure 10 shows the three ways in which the branch delay slot can be scheduled. Delayed branching is a simple and effective solution for the five-stage MIPS pipeline. However, as processors go to longer pipelines, the branch delay becomes longer and a single delay slot is not sufficient.

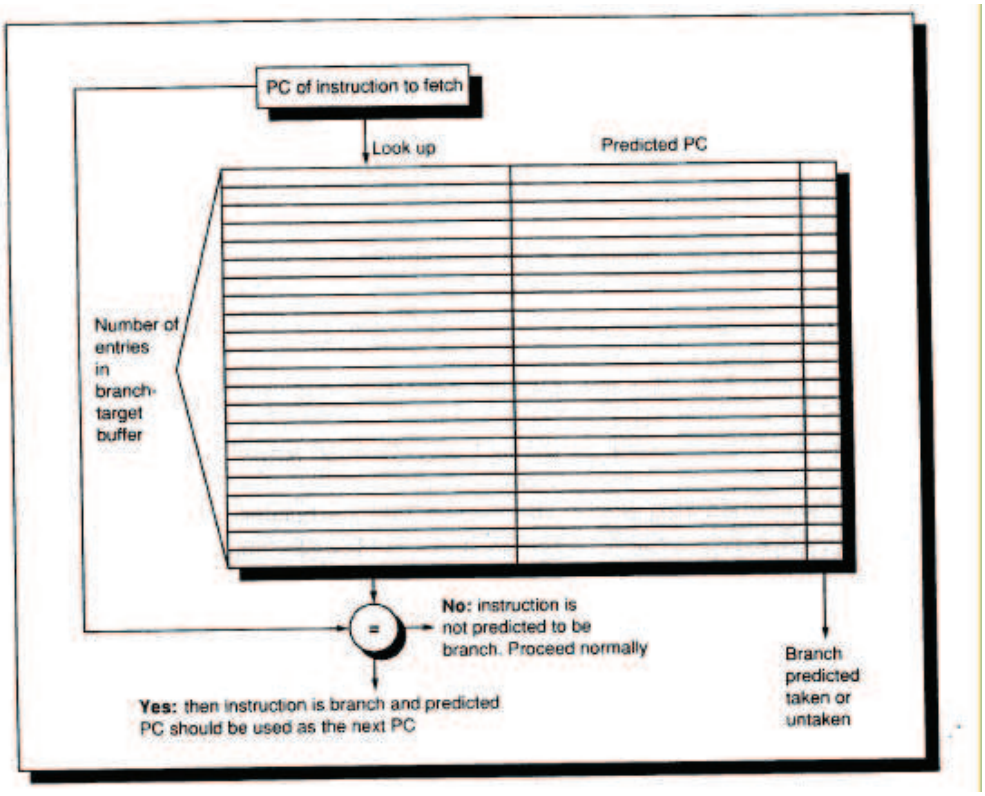


Figure 11: The branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction currently being fetched is a branch. If it is a branch, then the second column, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field just tracks whether the branch was predicted taken or untaken.

## Dynamic Branch Prediction in Hardware

To reduce the branch penalty in the MIPS pipeline, we need to know from what address to fetch by the end of IF. This means we must know whether the as yet decoded instruction is a branch, and if it is a branch, what the next PC should be. If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero. A branch-prediction cache that stores the predicted address for the next instruction after the branch is called a *branch-target buffer* (BTB). Because we are predicting the next instruction address and will send it out **before** decoding the current instruction, we **must** know if the fetched instruction is predicted as a taken branch. We also want to know if the address in the BTB is for a taken or not-taken prediction, so that we can reduce the time to find a mispredicted branch. Figure 11 shows the structure of a BTB. If the PC of the fetched instruction matches a PC in the buffer, then the corresponding predicted PC is used as the next PC.

If a matching entry is found in the BTB, fetching begins immediately at the predicted PC. Note that the entry must be for this instruction, because the predicted PC will be sent out before it is known whether this instruction is even a branch. If we did not check the whether the entry matched this PC, then the wrong PC would be sent out for instructions that were not branches.

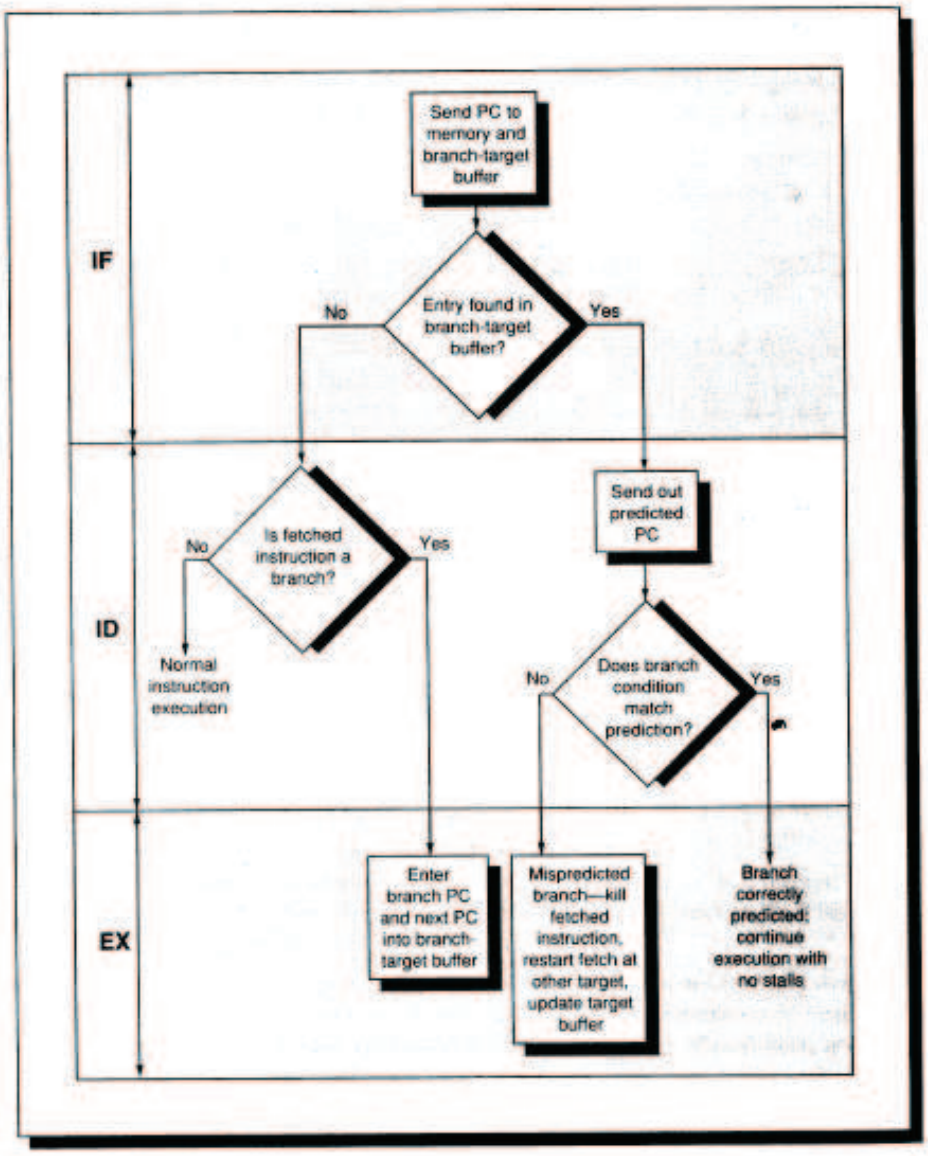


Figure 12: The steps involved in handling an instruction using a branch-target buffer.

Figure 12 shows a flowchart detailing the steps followed when using the BTB. If the PC of an instruction is found in the BTB, then it must be a branch and the fetching immediately begins from the predicted PC. If the entry is not found and it subsequently turns out to be a branch, it is entered in the BTB along with the target, which is known by the end of ID. If the instruction is a branch, is found in the BTB, and is correctly predicted, then execution proceeds with no delays. If the prediction is incorrect, we suffer a one-clock-cycle delay fetching the wrong instruction and a one-cycle delay updating the BTB, and restart the fetch. If the branch is not found in the buffer and the instruction turns out to be a branch, we will have proceeded as if the instruction were a branch and can turn this into an assume-not-taken strategy; the penalty will differ depending on whether the branch is actually taken or not.

To evaluate how well the BTB works, we must first determine what the penalties are in all possible

| Instruction in buffer | Prediction | Actual branch | Penalty cycles |
|---|---|---|---|
| Yes | Taken | Taken | 0 |
| Yes | Taken | Not taken | 2 |
| Yes | Not taken | Not taken | 0 |
| Yes | Not taken | Taken | 2 |
| No | | Taken | 2 |
| No | | Not taken | 1 |

Figure 13: Penalties associated with a BTB, depending on all possible combinations of whether the branch is in the BTB, how it is predicted, and what it actually does.

cases, and Fig. 13. There is no branch penalty if everything is correctly predicted and the branch is found in the BTB. If the branch is not correctly predicted, the penalty is one clock cycle to update the BTB with the correct information (during which time no new instruction can be fetched), and one clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found in the BTB, and is not taken, the penalty is only one clock cycle since the pipeline always assumes not taken, and is not yet aware that the instruction is a branch. Other mismatches cost two clock cycles, since we must restart fetch and update the BTB.

Let us assume that there is a 90% chance of finding a branch entry in the BTB, and if the branch is found in the BTB, there is 90% chance that we will predict that branch correctly. Also, assume that 60% of the branches in the program are actually taken. Then, the branch penalty due to the use of the BTB is

$$\begin{aligned}\text{Branch penalty} = {} & \%\text{ branches found in BTB} \times \%\text{ incorrect predictions} \times 2 + \\ & (1 - \%\text{ branches found in BTB}) \times \%\text{ taken branches} \times 2 + \\ & (1 - \%\text{ branches found in BTB}) \times \%\text{ untaken branches} \times 1 +\end{aligned}$$

So, the branch penalty is $90\% \times 10\% \times 2 + 10\% \times 60\% \times 2 + 10\% \times 40\% \times 1 = 0.34$ clock cycles. This compares well with a branch penalty of about 0.5 clock cycles per branch for the delayed branching concept. Recall that, typically, the compiler is able to fill only 50% of the delay slots. Also, the improvement from dynamic branch prediction methods will grow as the branch delay grows.